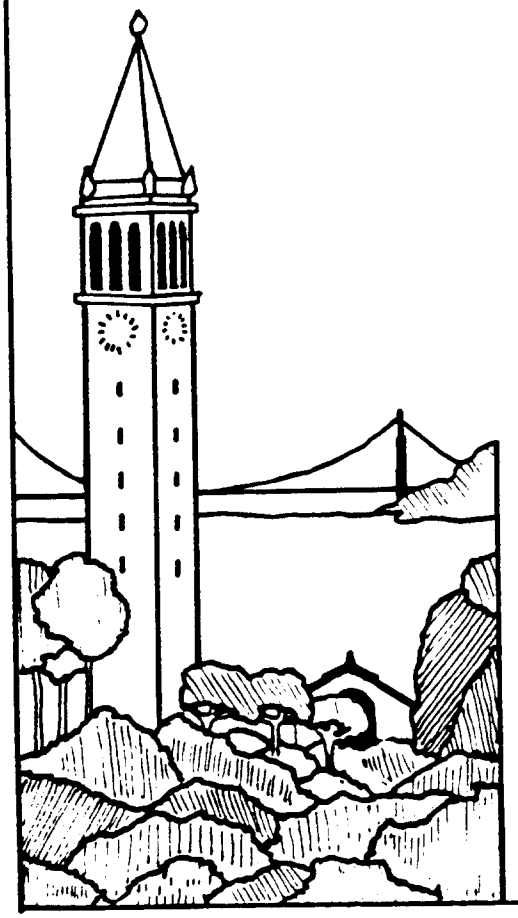
A Browser for Directed Graphs

Lawrence A. Rowe, Michael Davis, Eli Messinger,
Carl Meyer, Charles Spirakis, and Allen Tuan



Report No. UCB/CSD 86/292

April 1986

Computer Science Division (EECS)
University of California
Berkeley, California 94720

A Browser for Directed Graphs †

(Draft printed: March 25, 1986)

Lawrence A. Rowe, Michael Davis,[1] *Eli Messinger,*
Carl Meyer,[2] *Charles Spirakis,*[3] *and Allen Tuan*

Computer Science Division - EECS Department
University of California
Berkeley, CA 94720

Abstract

A general-purpose browser for directed graphs is described. The browser provides operations to examine and edit graphs. An operation is also provided to generate a layout for a graph automatically that minimizes edge crossings. Two layout algorithms were implemented. A hierarchical graph layout algorithm was found to be best for directed graphs.

The graph browser also has facilities that allow it to be integrated with other applications (e.g., a program browser or a database design tool). These facilities and our experiences building a program call-graph browser are described.

1. Introduction

The availability of low-cost personal workstations with a high resolution bit-mapped display and a mouse has made possible wide spread use of sophisticated editors for graphically displayed data. Data can be displayed in many different representations including: icons, pictures, typeset text, or forms [Cat80, Goe85, Her80, StK82, RTI84, Zlo75]. These editors are often called *browsers* because users navigate through the data, examining and changing it.

A directed or undirected graph is a good way to display complex relationships and data dependencies. For example, graphs can represent the module dependency relationship in a large software system, the call-graph in a program,

† This research was sponsored by the Defense Advanced Research Projects Agency (DoD) under Arpa Order No. 4871 and monitored by the Naval Electronic Systems Command under Contract No. N00039-84-C-0089.

[1] Current address: SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025.
[2] Current address: Sun Microsystems, Inc., 2550 Garcia Ave., Mountain View, CA 94043.
[3] Current address: Tandem Computers, 19333 Vallco Parkway, Cupertino, CA 95014.

the finite state automaton underlying an LR parser, or a database design. This paper describes the design and implementation of a graph browser, called GRAB, that can be used to browse arbitrary directed graphs or that can be integrated with an application-specific browser (e.g., a program browser or a database design tool). The remainder of this section presents an overview of GRAB and discusses related research on browsers and automatic layout algorithms.

GRAB displays the nodes of a graph as icons and the edges as lines drawn between the icons that represent its endpoints. Figure 1 shows a call-graph for a small program displayed by GRAB. The nodes of the graph represent the procedures and the edges represent the CALLS relationship. Notice that the edges are directed to indicate the calling procedure and the called procedure. In this example, all nodes represent the same type of data, namely a procedure, so they are represented by the same icon, a box, labeled with the name of the procedure. Different node types can be displayed as different icons and labels can be

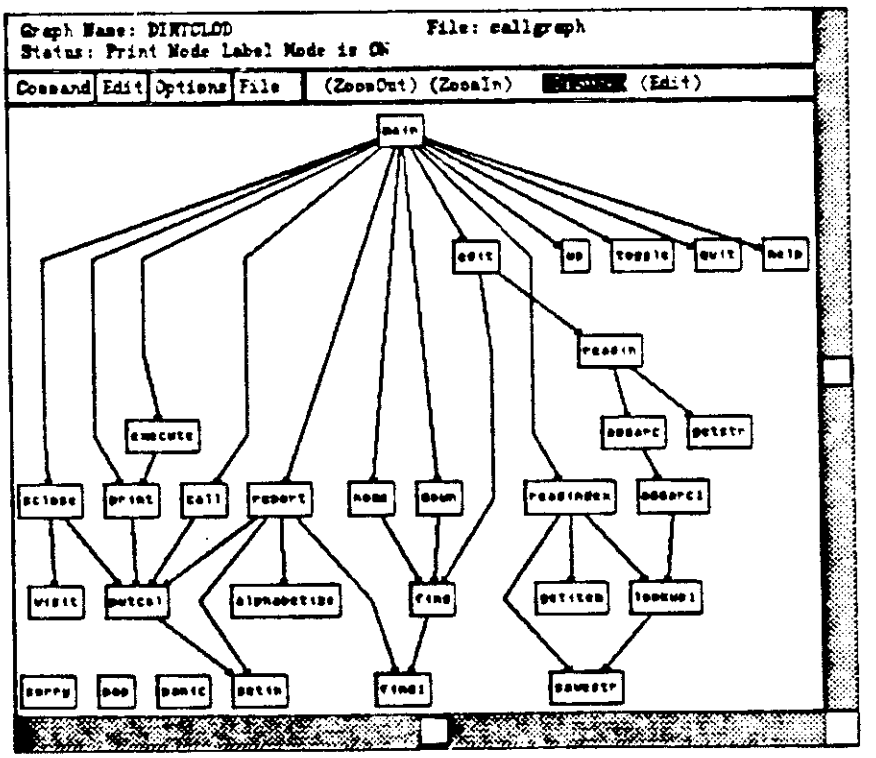


Figure 1. A call-graph displayed by GRAB.

displayed on the edges (e.g., the number of calls made from one procedure to another).

The user can examine a graph by scrolling horizontally and vertically (i.e., panning) and by zooming in on one part of a graph. Figure 2 shows the graph in figure 1 after the user has zoomed in on the "report" procedure. GRAB also provides editing operations that allow the user to insert or delete nodes and edges, to move nodes, and to change a node or edge label.

Most browsers that display information in a graph require the user to specify the graph layout manually. Manual layout is tedious, time-consuming, and often produces a poor representation of the information. GRAB provides an operation to lay out a graph automatically. The objective of the layout algorithm is to minimize the number of edge crossings and to position nodes connected by an edge close to one another.

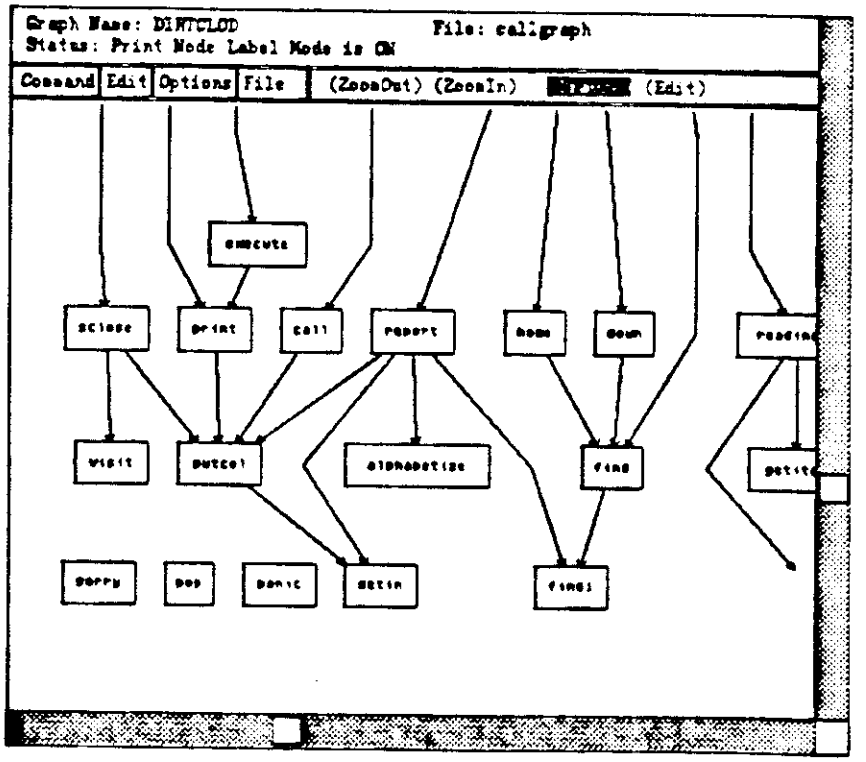


Figure 2. Display after the user has zoomed in on a subgraph.

In addition to these browsing and editing operations, a user can invoke an entity-specific browser for a node that opens a separate window through which detailed information about the node can be displayed. For example, suppose the entity-specific browser for the call-graph shown in figure 1 is a text editor that displays the source code of the selected procedure. Figure 3 shows the screen after the user has invoked the entity-specific browser on the "report" procedure node. The user can browse the source code and the call-graph and he can rapidly switch to the code for a different procedure.

The design of GRAB was influenced by research in three areas: 1) application-specific editors, 2) entity-browsers, and 3) automatic layout algorithms. Numerous editors that display information in a graph have been built for applications like project scheduling [APP84d], database design [BrH86], user-interface specifications [MiW84], and circuit design [SCA83]. These editors typically use a fixed set of icons for the graph nodes, they impose other constraints on

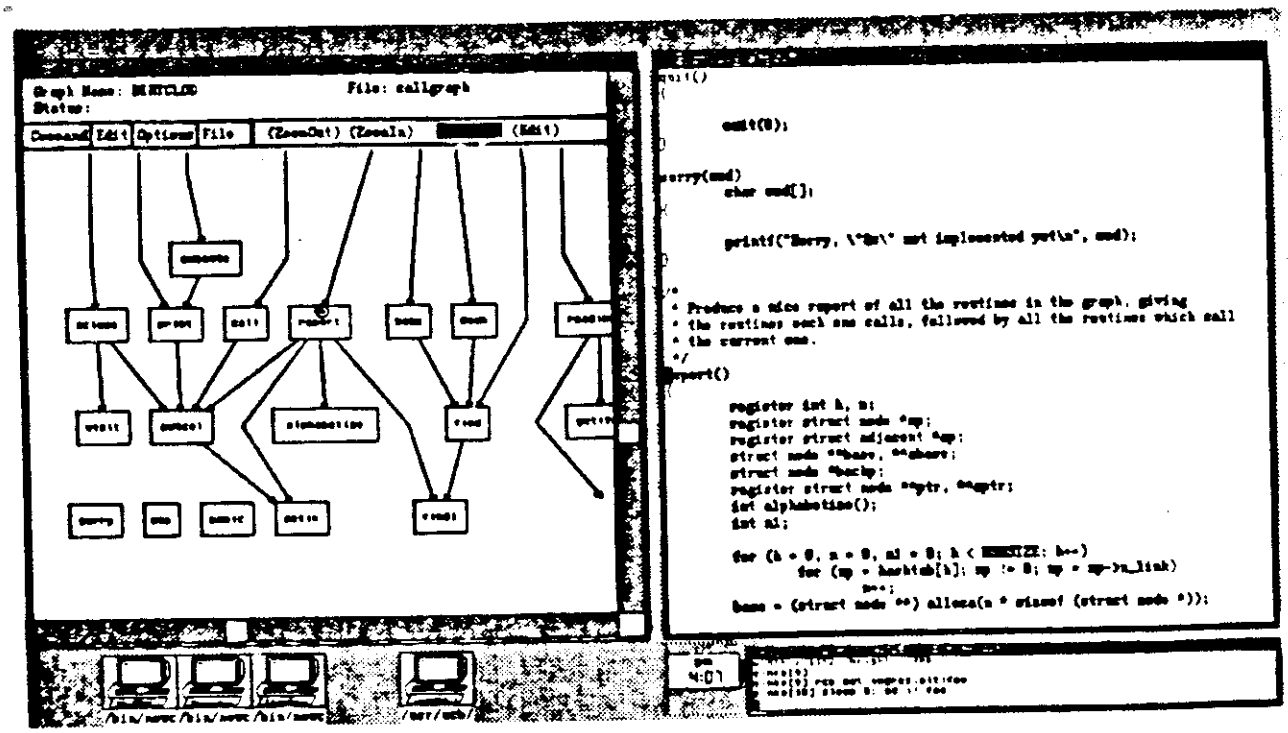


Figure 3. Example of invoking a node-specific browser.

the graph layout related to the specific application, and they do not provide an automatic layout operation. For example, a graphical display of an entity-relationship database design uses two icons: a rectangle for entities and a diamond for relationships [Che76] as shown in figure 4 [Tae83]. Two constraints are imposed on this layout. First, edges are composed of segments that are perpendicular to the x- or y-axis (i.e., manhattan geometry). Second, the edges are connected to the icons at particular points (e.g., edges connect to a diamond on one of the diamond's points). This example illustrates some of the constraints that a complete graph browser tool has to provide.

Previous work on entity-browsers [Cat80,Her80,StK82] has not focused on displaying graphs. Systems that do display graphs have trivial layout algorithms. For example, Cattell's entity-browser lays out the nodes of a graph by spiraling out from the center until an unfilled position is found where the next node is placed. The heuristic makes no attempt to reduce the number of edge crossings or even to reroute edges that intersect other icons.

Several algorithms have been developed that automatically lay out graphs [Lie85,Mar85,STT81,Tae83,Woo81]. We have implemented and experimented with the algorithms developed by Sugiyama, et.al. [STT81] and Woods [Woo81]. Our objective was to find an efficient algorithm that would produce reasonable looking layouts for arbitrary directed graphs. In addition, the algorithm should allow the user to define constraints on the layout (e.g., put this node at the top and lay out the graph hierarchically down the page).

GRAB currently uses a variant of Sugiyama's algorithm that was originally designed to lay out directed graphs with no cycles.[1] We chose this algorithm over Woods' algorithm, originally designed to lay out undirected planar graphs, because we could extend it to meet more of our objectives. The Sugiyama algorithm was modified to handle cycles and to improve the layouts. The major problem with Woods' algorithm was that we could not make it produce hierarchical layouts for directed graphs.

Wire routing in VLSI chips is a related problem that has received considerable attention recently [BuP83,HaO84,Hsu83,Sae84]. While these algorithms could be adapted to produce layouts that minimize the number of edge crossings, we did not think they would work well in our application because we need a fast algorithm that will produce reasonable layouts. We are willing to sacrifice layout quality for run-time performance. In contrast, layout quality is more important than run-time performance in wire routing because the quality of the routing directly effects the space/time performance of the resulting chip. We investigated several wire routing algorithms and found that they ran slower than the other algorithms investigated.

[1] The original algorithm collapsed cycles into a single node.

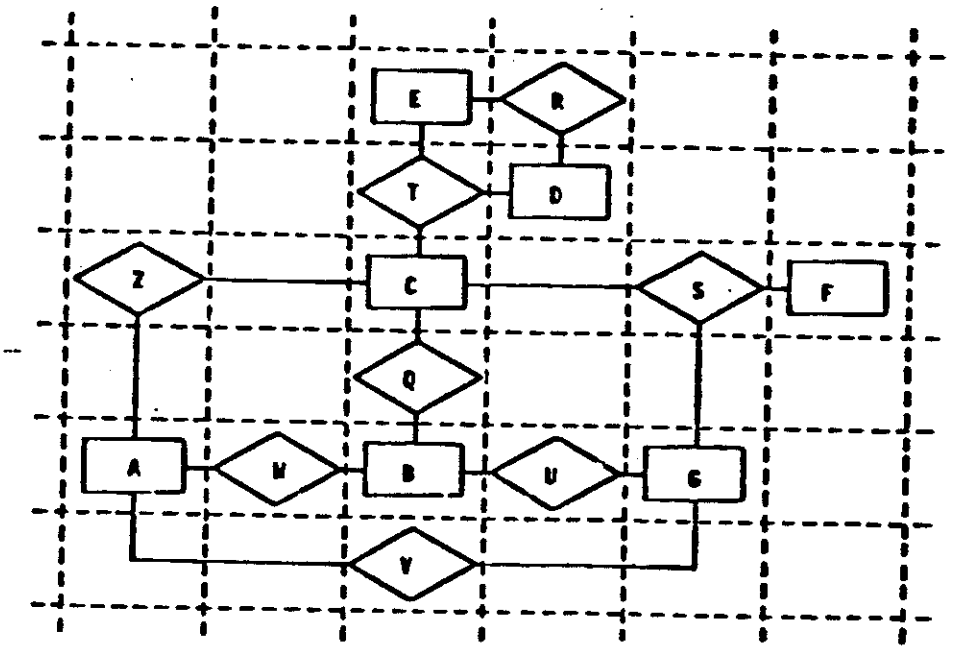


Figure 4. An entity-relationship database design graph.

This paper describes the extensions we made to Sugiyama's graph layout algorithm and the experiences we had building the program browser illustrated above. The remainder of the paper is organized as follows. Section 2 describes the Sugiyama algorithm including the improvements we made to it. Section 2 also describes our attempt to extend Woods' algorithm. Section 3 describes the program browser and the problems we encountered using GRAB to implement it. Section 4 describes rules that we learned during the course of this project about the design of menu-based, graphical applications. Lastly, section 5 summarizes the paper.

2. Graph Layout Algorithms

This section describes our implementation of Sugiyama's algorithm, the improvements we made to it, and our attempt to extend Woods' algorithm.

The nomenclature used in this section is illustrated by the example in figure 5. A *node* and an *edge* are defined in the usual way for directed graphs. The nodes that can be reached from a given node by following the directed edges leaving the node are called the *successors* of the node. The nodes from which a given node can be reached by following directed edges are called the *predecessors* of the node.

A *level* is a discrete up/down position. They are numbered from top to bottom, starting with level 0 at the top of the graph. In a strict hierarchy, each node will have all ancestors at lower levels and all descendents at higher levels.

Edges that span more than one level are called *long edges*. A *dummy node* is added at each intermediate level to indicate where the edge crosses the level. A long edge can change direction only at the positions of the dummy nodes. A dummy node has no size or label but is otherwise treated like a normal node by

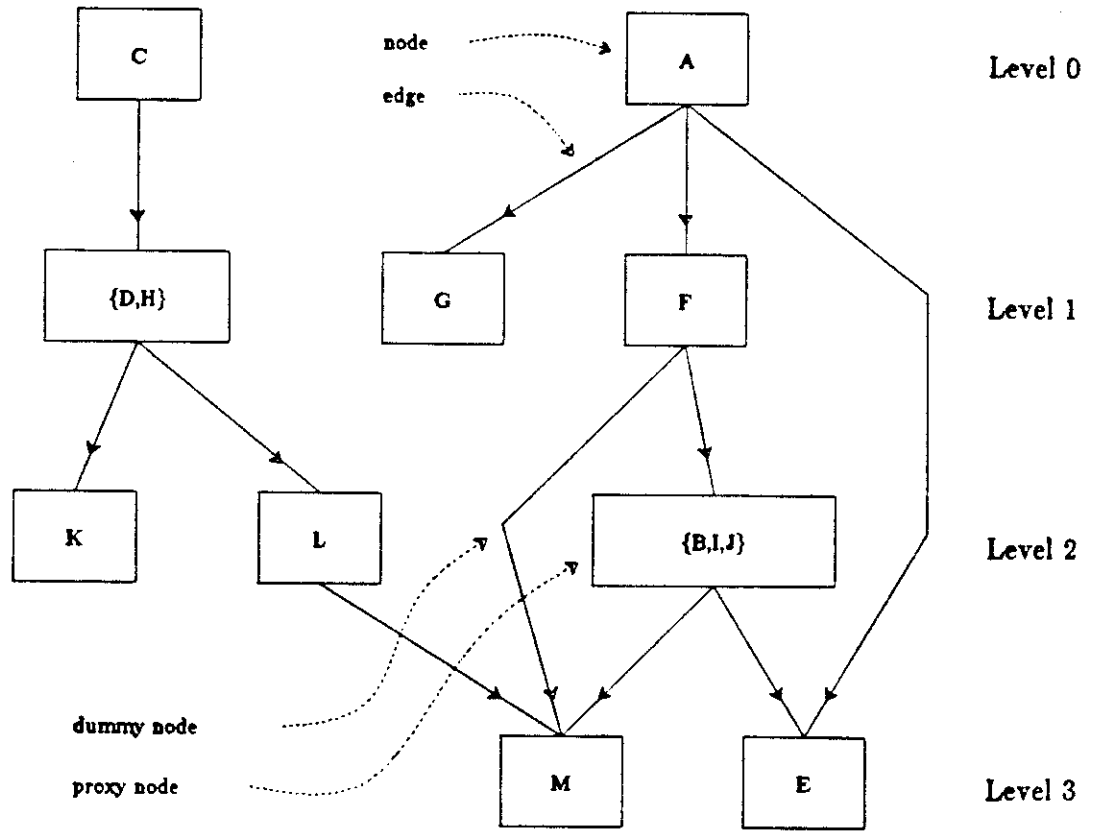


Figure 5. Nomenclature for layout algorithm.

the layout algorithm.

A *cycle* is a set of nodes such that there is a closed directed path from each node in the set to itself that passes through the other nodes in the set. It is not possible to lay out the nodes in a cycle hierarchically, so the original Sugiyama algorithm collapsed all the members of a cycle into a single node, called a *proxy*. The algorithm treated a proxy as a single node and labeled it with the names of all the nodes in the cycle. In the figure, {B,I,J} is a proxy replacing the cycle B→I→J→B.

The remainder of this section describes the algorithm currently used in GRAB. We also describe how we solved the problems we ran into with the original algorithm and the changes that could be made to further improve the algorithm. The last subsection describes the problems we encountered while attempting to extend Woods' algorithm to nonplanar directed graphs.

2.1. A Hierarchical Graph Layout Algorithm

The hierarchical layout algorithm developed by Sugiyama et.al. [STT81] has three phases. The first phase assigns nodes to levels. The second phase sorts the nodes on each level to minimize the number of edge crossings. The third and final phase fine tunes the positioning of nodes and routing of edges to make the layout easier to understand. Each of these phases is discussed in turn.

Nodes are assigned to levels in the first phase so that each node is on a level below its ancestors. The first step is to compute the transitive closure of all descendents of a node to determine which nodes have no descendents and therefore belong at the bottom of the graph. Nodes are assigned to levels by assigning nodes with no descendents to the current level, removing them and their edges from the closure, and performing the same operation on the next level up the graph until all nodes have been assigned to a level.

The second step to assigning levels is to break up long edges that connect nodes that are more than one level apart. These long edges are broken into segments, each of which goes between adjacent levels, and dummy nodes are inserted at the intervening levels. Dummy nodes and the segments of a long edge are treated by the algorithm as normal nodes and edges. The reason for segmenting the edges and creating the dummy nodes is to allow edges to bend at each level, thereby providing flexibility in positioning of the two nodes connected to the long edge and permitting a more compact graph layout.

As mentioned above, the original algorithm did not lay out cycles. A simple heuristic was added to solve this problem. During the transitive closure step, the descendents of a given node are found by doing a depth-first search. Whenever a node is found to be its own descendent, a cycle is indicated. Cycles are "broken" by temporarily reversing the edge that completed the cycle and proceeding with the algorithm. When the graph is displayed, the reversed edge is drawn in the reverse direction (i.e., in the correct direction). The nodes connected by the edge appear on the levels that they would have occupied if the edge had been directed

downwards.

Figure 6 shows the layout of the call-graph after the completion of phase one of the algorithm. The nodes have been assigned to levels and long edges have been broken up, but the nodes on each level have not yet been sorted.

The second phase of the algorithm makes multiple passes over the graph changing the order of nodes on each level to minimize the number of edge crossings. The first pass begins at the top level of the graph and moves down. The second pass begins at the bottom level and moves up. Subsequent passes alternate between top-down and bottom-up passes until the algorithm terminates.

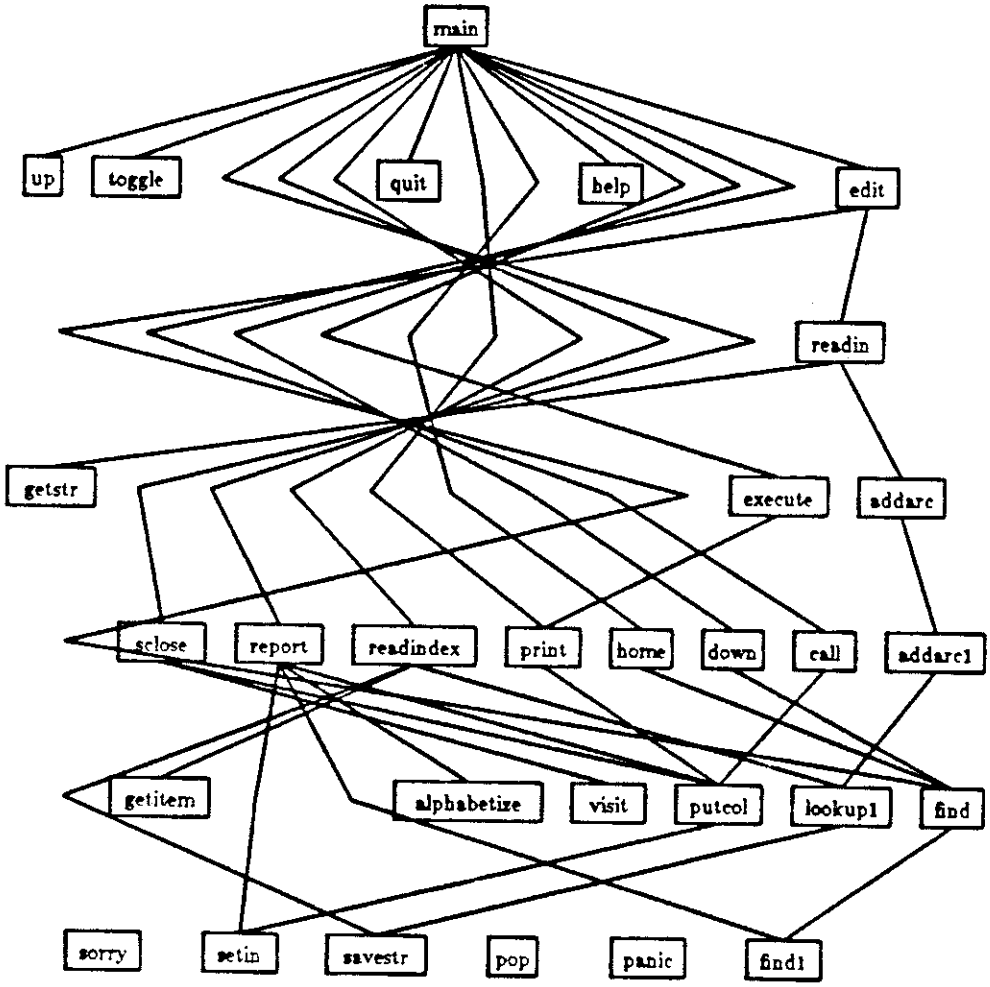


Figure 6. Call-graph after the level assignment phase.

A number that estimates the best horizontal position of each node, called a *barycenter*, is used to order the nodes on a level. The original algorithm used two barycenters for each node: the *up-barycenter* and the *down-barycenter*. The up-barycenter (down-barycenter) of a node is the average position of its immediate predecessors (successors) including dummy nodes on the previous (next) level. Sorting the nodes on a level by up-barycenters (down-barycenters) comes close to minimizing the number of edge crossings with the previous (next) level.

The original algorithm sorted nodes on a level by up-barycenters when making a downward pass and by down-barycenters when making an upward pass. This approach caused node positions to be unstable in some graphs. If the position of a node determined by down-barycenters was very different than the position determined by up-barycenters, the node would be moved back and forth between the two positions on alternate passes. Often, the position leading to the smallest number of crossings was between the two extremes.

We solved this problem by changing the position estimating function after the first pass downward and upward. The first pass (downward) sorts by up-barycenter. The second pass (upward) sorts by down-barycenter. The third and subsequent passes sort by the average of the up- and down-barycenters of a node. This solution has worked reasonably well on the graphs that we have laid out.

The second phase of the algorithm terminates when all edge crossings have been eliminated or after a fixed number of passes have been made over the graph (typically 4 or 5). Figure 7 shows the call-graph after the second phase. Notice that the number of crossings has been reduced, but that the jagged edges and unbalanced positioning of the nodes makes the graph hard to read.

The third phase of the algorithm fine tunes the layout. The earlier phases determine the level assignment and the order of nodes on a level. The fine tuning phase determines the actual xy-coordinates for each node. This phase is composed of several steps. First, the nodes are evenly distributed on a level and the levels are separated by a uniform distance. This step takes into account that nodes can have different heights and widths depending on the icon that represents it.

After the nodes and levels are positioned, the second step makes several passes over the graph to straighten long edges. Dummy nodes are moved to be in line with their predecessor and successor nodes (i.e., to straighten long edges). Recall that a dummy node has only one predecessor and successor. This heuristic changes the position of dummy nodes on a level but it will not change the order of nodes on a level.

The final step adds spacing between levels so that the slope of any edge between two levels is at least 2/3. This heuristic may enlarge the diameter of the graph but it eliminates tightly packed levels with nearly horizontal edges.

The final layout of the call-graph is shown in figure 1. As can be seen by comparing figures 1 and 7, fine tuning improves the readability of a graph. More

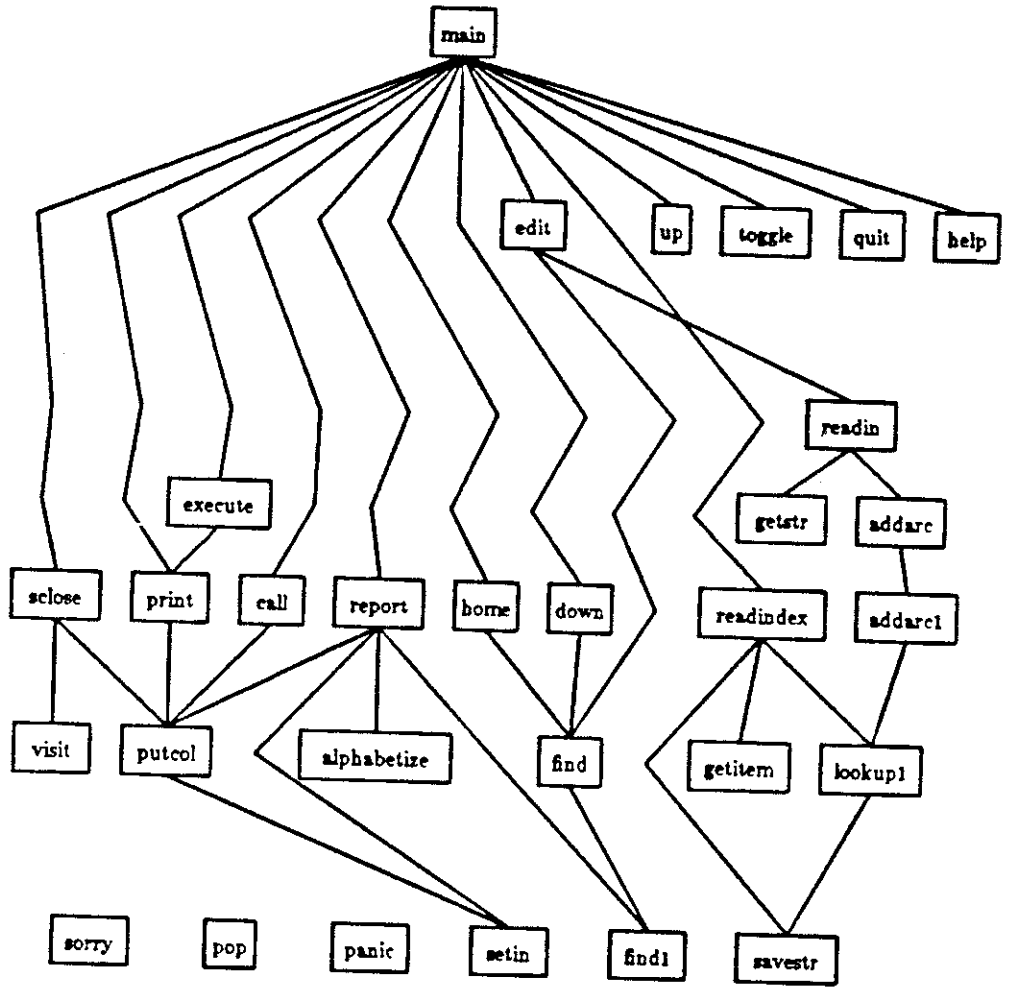


Figure 7. Call-graph after the sorting phase.

details on the implementation are given elsewhere [Dav84,Mey83].

2.2. Further Improvements

This section compares layouts produced by the current GRAB algorithm to layouts produced by the original Sugiyama algorithm and describes two improvements that still need to be made.

The paper describing the Sugiyama algorithm showed a layout of a graph representing a dynamic world model developed by Forrester [For71]. The layout had 72 edge crossings. Because their algorithm did not handle cycles, a node in

each cycle was replicated to eliminate the cycle. Figure 8 shows the layout produced by our algorithm when the cycles were left in the graph. This layout has only 49 crossings which is 30% fewer than in the Sugiyama layout.

Our current implementation has two problems that need to be fixed. The first problem is with the placement of clusters of nodes at the end of a series of long edges. For example, nodes 14, 39, 20, and 15 in the lower left corner of figure 8 could be moved up higher in the graph. The level placement phase of the algorithm needs to be modified to identify clusters of nodes that can be moved higher in the layout to reduce total edge length. Sugiyama independently discovered this same idea, and devised a heuristic [Sug84].

The second problem occurs in the crossing minimization phase. The current algorithm does not eliminate certain edge crossings because the level-sorting heuristic looks only at pairs of adjacent levels. For example, the edge from node 30 to 33 in figure 8 crosses two edges. This number could be reduced to one by placing node 33 to the right of the edge between nodes 29 and 34. The current algorithm misses this improvement because it only uses local information (i.e., the positions of immediate predecessors and successors) when sorting nodes on a level. The heuristic looks for a local minimum but not a global minimum. We have experimented with "weighted average" metrics that would take more levels into account, but to date, none has performed better than the current metric that averages the up- and down-barycenters.

This subsection compared our extended algorithm to the original Sugiyama algorithm and discussed known problems with the current implementation.

2.3. A Planar Graph Layout Algorithm

This section presents an overview of Woods' algorithm for laying out undirected planar graphs [Woo81] and our attempt to extend it to produce hierarchical layouts for directed nonplanar graphs. The Woods' algorithm is similar to the Sugiyama algorithm in that it is also divided into three phases.

The first phase, called the *planarisation phase*, produces an abstract description of a set of *planar embeddings*. Each embedding is an assignment of nodes and edges to positions in a plane such that no edges in the graph cross. The planarisation is produced by Hopcroft and Tarjan's linear algorithm for testing planarity [HoT74].[2] The second phase uses the planarisation to lay out the graph on a plane. The third phase fine tunes the layout using heuristics that are similar to the heuristics used in the fine tuning phase of the Sugiyama algorithm.

We implemented phases one and two of Woods' algorithm and modified them to lay out nonplanar graphs. The planarisation phase was modified to produce the abstract representation of a "planarisation" even though the graph was

[2] The algorithm presented in their paper contains several errors. A better presentation is given in Reingold, et.al.'s book on combinatorial algorithms [Ree77].

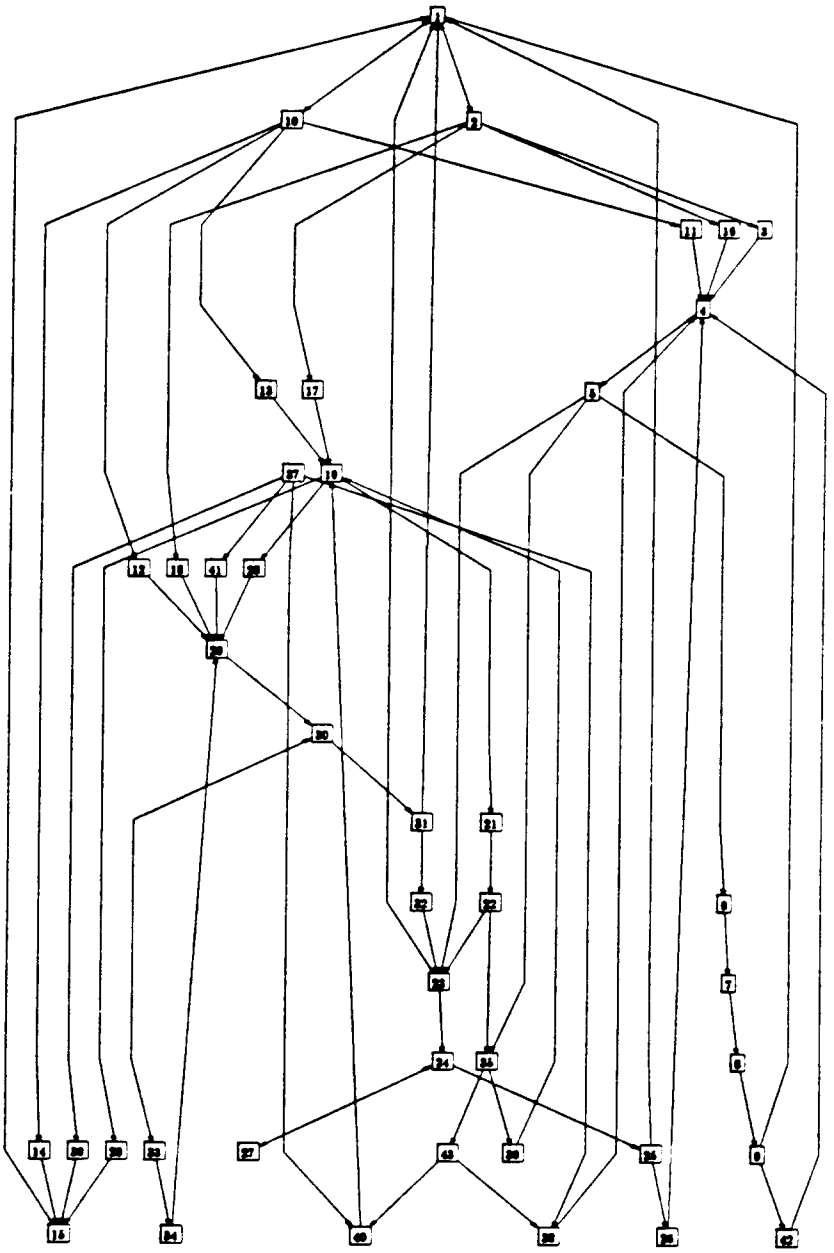
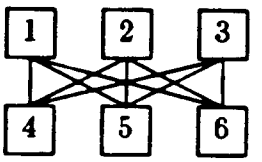


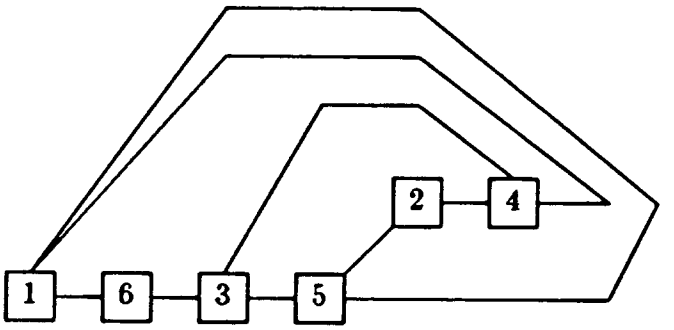
Figure 8. World dynamic model layout.

nonplanar. For example, suppose we wanted to lay out the graph K(3,3) shown in figure 9(a), which is known to be nonplanar. If the planarity algorithm divides the graph into an initial cycle 1-6-3-5-1 and subsequent paths 5-2-4-1, 4-3, and 2-6, the last path (i.e., 2-6) cannot be inserted without introducing a crossing as illustrated in Figure 9(b). Our solution was to insert the edge randomly and to ignore the resulting crossings.

The second phase was modified to generate a layout of the graph regardless of the consequences. The layouts produced did not look very good in part because we were not able to generalize an important heuristic that compacts the graph. The second phase of Woods' algorithm places few nodes on each level and depends on a compaction heuristic to reduce the number of levels required. The heuristic moves nodes on adjacent levels to the same level if a horizontal edge can



9(a). K(3,3)



9(b). "Planarisation" before inserting 2-6.

Figure 9. "Planarisation" of K(3,3).

be drawn between them without intersecting another node. Without compaction, the layouts are long and narrow. We could not use Woods' compaction heuristic because it depended on the graph being planar and properties of the abstract representation of the planarisation. Since our graphs were not planar, the heuristic could not use the abstract representation to decide when levels could be eliminated. Because we could not solve the hierarchical placement problem described in the next paragraph, we abandoned our attempts to find a new compaction heuristic.

We discovered another problem with this algorithm as we were experimenting with it. Since the algorithm ignores the direction of edges when deciding where to place nodes, edges were randomly directed. In other words, the layouts did not even come close to approximating a hierarchical layout where the majority of edges point in one direction. When we started this project we did not realize the importance of producing hierarchical layouts. Our experience looking at layouts produced by this algorithm and large graphs displayed by the program browser described below has convinced us of the importance of generating hierarchical layouts. They are easier to understand. We explored modifications to the algorithm to solve this problem, but we could not find a solution.

Our current thinking is that for directed graphs, only graphs that can be laid out compactly by taking advantage of symmetric structure in the graph should be displayed non-hierarchically. A good example of a layout algorithm based on symmetric structure is described by Lipton [Lie85].

3. A Program Browser

This section describes our experience building and using the program browser described in the introduction. The browser was relatively easy to implement because an existing cross-reference program could be used to generate the call-graph.[3] We used the program browser on several different programs which uncovered problems with the browser itself and the tools used to build it.

The biggest problem was that the layout algorithm ran too slowly on large graphs. For example, it took 13 minutes on a Sun-II with a Sky floating point board [SUN84] to lay out and display the call-graph for the user-interface to GRAB (approximately 9000 lines of code and 280 functions). A much faster algorithm is needed to make the graph browser practical for large graphs.

As a result of this experience, we are working on a new, faster algorithm. The algorithm will decompose a graph into a collection of subgraphs. The subgraphs will be laid out separately and edges that connect nodes in different subgraphs will be routed through the graph. We will use our extended version of the Sugiyama algorithm to layout the subgraphs. To make combining subgraphs and routing edges between subgraphs easier, we are adding constraints to the basic

[3] The GRAB input format for graph data is a list of nodes followed by an edge list.

layout algorithm, such as "position a particular node on the right-hand edge of the subgraph" and "position a node above another node". The advantage of the decomposition approach is that it will reduce the problem size without hurting the quality of layouts. Another advantage is that it may be possible to use multiprocessor machines to lay out the subgraphs in parallel.

The second problem we discovered was that the layout of the graph should remain stable even though nodes and edges have been inserted and deleted as a result of changes to the program being browsed. If the layout remains stable, the user can retain a model of the general structure of the graph and the relative position of nodes. At present, GRAB arbitrarily repositions the nodes in a graph when it lays out the graph after it has been changed. Our current thinking is that as nodes are added to the layout they can be heuristically placed by GRAB until one of the following situations causes the entire graph or one or more subgraphs to be laid out again:

1. a particular subgraph becomes too complicated (i.e., too many edge crossings are introduced or there are too many "cross subgraph" edges),
2. the subgraph partition must be changed (e.g., a set of nodes should be moved to another subgraph or a subgraph should be partitioned), or
3. the user explicitly requests a new layout.

Another problem exposed by our attempts to browse large call-graphs was that the graphs were difficult to read because of the large number of crossings and the high density of edges following a similar route. We made changes to the call-graph browser and GRAB in an attempt to solve this problem. First, we simplified the call-graphs by eliminating nodes for common utility routines (e.g., input/output routines). Second, we added operations to GRAB to highlight the edges exiting from or entering a specific node. Figure 10 shows the call-graph from figure 1 with the edges exiting from the routine "report" highlighted. Edges can be highlighted interactively by selecting a node and executing an operation to highlight "entering," "exiting," or "entering and exiting" edges. While both changes made the graph easier to understand, they did not solve the problem. More work is required on visual representations and browsers for large programs.

The program browser was also interfaced to the *vi* text editor [Joy84]. The user could invoke the editor on the source code for a particular routine by positioning the mouse on the node and pushing a button. The editor was passed the name of the routine and it used a built-in facility to select the correct file to edit and to position the user at the beginning of the appropriate definition (*ctags*). An example of this operation was shown in figure 3.

Although a simple version of this operation was implemented, it provided good feedback on how to improve our tools. First, we have to allow changes made to the source code through the editor to be propagated back to the graph

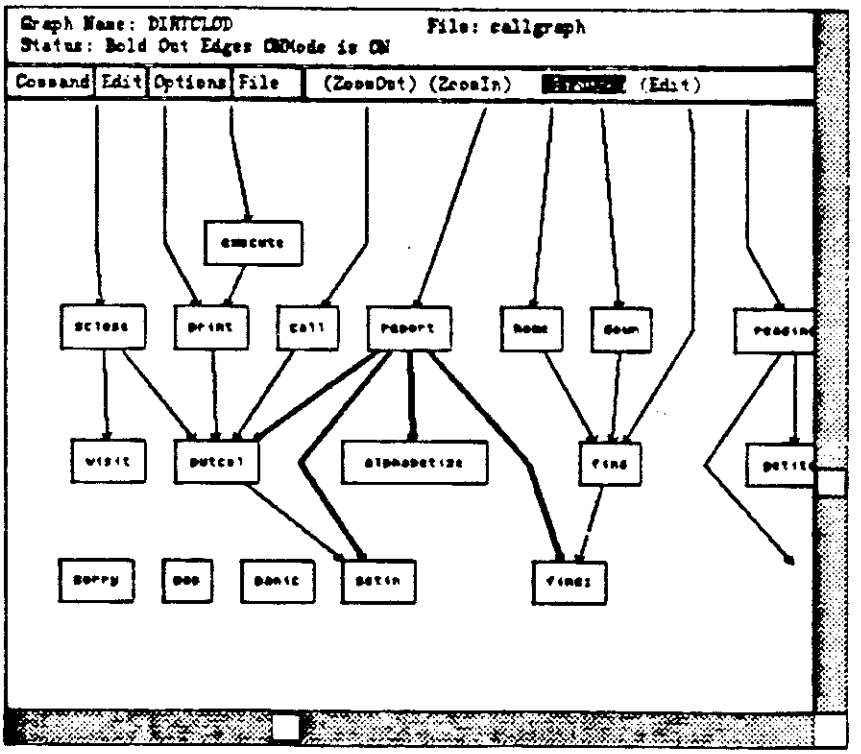


Figure 10. Program call-graph with edges highlighted.

representation. We did not implement this feature in the prototype because it would have required significant modifications to the editor. We have designed and implemented the required protocol for GRAB [Row85] and are integrating it into a language-independent structured editor being developed by another research group at Berkeley [Bal85].

Another problem with the current implementation is that two editors can be spawned on the same file. A concurrency control mechanism is needed that will allow only one user at a time to update the source code. One solution is to store the source code and the graph representation in a database management system (DBMS). This implementation would allow concurrent browsing and updating of both the program and the graph. In addition, larger graphs could be browsed because the graph data structure would not have to be kept in main memory as is done in the current implementation. We would have implemented the current version of GRAB on a DBMS but a conventional DBMS is not fast enough to provide the real-time response required. The INGRES project at Berkeley has begun

the development of a new DBMS that will have facilities that will make it possible to store the graph data structure in a database. The special facilities include: geometric data types, spatial indexes, and compiled queries [StR85].

Lastly, GRAB should be changed to make it easier to customize the system for a specific browser application. For example, it should be possible to limit the user to selected operations (e.g., only allow the "move node" editing operation) if the other editing operations are not meaningful in a particular application. It should also be possible to spawn a non-GRAB operation when a GRAB operation is executed. For example, suppose you were building a browser for a hierarchical file system that displayed the hierarchy as a directed graph. Then, if the user executed an operation to delete a node in the graph, the browser could spawn a "delete file" operation to the file system.

This section described our experiences developing a program browser based on GRAB.

4. Designing the User-Interface

This section describes what we learned about the design of menu-based, graphical applications while developing GRAB. These concepts will be expressed in the form of "rules for a designer."

The first rule is to provide as much visual feedback as possible. We change the icon that denotes the mouse cursor to indicate when it is positioned on a node or an edge. This feedback tells the user when he has selected a particular object on the screen. This change made a dramatic impact on the usability of the interface. In an earlier version of GRAB that did not have this feature, it was hard to edit the graph because you could not determine when the node or edge was actually selected. Visual feedback through the mouse cursor solved the problem. However, it did require that we maintain a bin structure on the objects in the graph (i.e., a spatial index) to reduce the time required to test whether the cursor is positioned on an object.

The second rule concerns the use of modes in an interface. Some people argue that interfaces should be modeless [Tes81]. Unfortunately, you cannot avoid the introduction of modes but you can make a moded interface easier to understand and use. The design rule is to be cognizant of the modes you introduce and to make them appear "seamless." By seemless we mean that the modes do not appear to exist. The trick is to provide visual feedback that indicates the current mode, the available modes, and to make it easy to switch between modes. We used techniques similar to those used in the drawing programs on the Macintosh [APP84b,APP84c]. The current mode is indicated by changing the mouse cursor to an icon that indicates the mode. The available modes are represented by a palette of choices. The two modes in GRAB (i.e., "Browse" and "Edit") are represented by buttons on the right end of the menu bar (see figure 1). Switching between modes is accomplished by positioning the cursor on the desired mode in the palette and pushing a mouse button. This style of interface avoids the major

problem with moded interfaces on conventional alphanumeric terminals which is that the user gets lost because he cannot tell which mode he is in, what other modes are available, and how to switch modes. The key concept is to provide visual feedback.

The third rule of interface design is to provide the right operations. The user of your interface will have some conceptual model of the task he wants to perform using the system. That conceptual model includes high-level actions, call them "semantic actions," he wants to execute. The user must map these semantic actions to the operations provided by your interface in his head. If your interface provides operations that match the semantic actions, the interface will be easier to use, and it will be perceived as a powerful system. The big problem in designing user-interfaces is to determine what semantic actions to provide as operations.

During the development of GRAB we added and removed many operations. On several occasions operations were added in response to a problem we found when trying to use the system. For example, we introduced new operations that made it easier to change between different views of the graph. Suppose the user has zoomed in several steps and panned to one side of the graph, and he wanted to return to the top level view that shows the entire graph centered on the screen. Because it takes several seconds to implement this semantic action by executing the pan and zoom operations in reverse order, the interface was frustrating to use. We added an operation "display to fit" that redisplays the top level view of the graph in one step. This change made the interface much easier to use.

Other examples of operations we added to make the system more usable included:

1. "Focus node" – redisplay the graph with the selected node centered in the upper third of the screen and zoomed in to reveal the node label.
2. "Highlight edges" – bold the edges exiting or entering a selected node.
3. "Force labels" – display the labels even though they do not fit inside the icon.

This last operation has both a static and dynamic version. The static operation displays all labels. The dynamic operation displays the label of the currently selected node when in browse mode. The idea was to make it easier to browse large graphs with small icons, possibly even points. With small icons you could not determine what each node represented because there was not enough space to print the label. The "force labels" operation solved this problem.

Operations were removed when we found a way to coalesce similar operations or when an operation was no longer needed. For example, in an earlier version we had two delete operations: "delete node" and "delete edge." We eliminated one operation by making the "delete" operation remove the selected object which was either a node or an edge. Because the interface provided visual feedback as

to which object was selected, the user knows what semantic action is being invoked.

A rule that encourages the introduction of new operations might produce a complex interface if too many operations are provided. Our experience is that by the appropriate use of visual feedback and consistency between operations, it is possible to provide many operations and the user will not realize it. For example, on the Apple Macintosh every window has 13 built-in operations (e.g., "move," "resize," "close window" and five horizontal and five vertical scroll operations) [APP84a]. Most users are not cognizant of these operations because the system provides good visual feedback and there is consistency between all windows.

As often happens with user-interfaces, these rules seem obvious once you have read them, but we had not thought about them before we developed GRAB.

5. Summary

This paper described the design of a general-purpose browser for displaying information as a directed graphs. The novel feature of this browser is the inclusion of an operation to lay out a graph automatically. We implemented two layout algorithms developed by others and experimented with extensions to meet our objective to produce reasonable looking layouts in real-time.

The paper also described a browser for program call-graphs. This prototype browser uncovered several problems with the graph browser and with call-graph browsers for large programs. The major problems were the run-time performance of the layout algorithm and the difficulty of understanding call-graphs for large programs.

References

[APP84a] *Apple Macintosh User's Guide*, Apple Computer, Inc., Cupertino, CA, 1984.

[APP84b] *MacPaint*, Apple Computer, Inc., Cupertino, CA, 1984.

[APP84c] *MacDraw*, Apple Computer, Inc., Cupertino, CA, 1984.

[APP84d] *MacProject*, Apple Computer, Inc., Cupertino, CA, 1984.

[Bal85] R. Ballance, personal communication, Dec. 1985.

[BrH86] D. Bryce and R. Hull, "SNAP: A Graphics-based Schema Manager", *Proc. IEEE 2nd Int. Conf. on Data Eng.*, 1986.

[BuP83] M. Burstein and R. Pelavin, "Hierarchical Wire Routing", *IEEE Trans. on CAD*, Oct. 1983, 223-234..

[Cat80] R. R. G. Cattell, "An Entity-Based User Interface", *Proc. 1980 ACM SIGMOD Conf. on Mgt. of Data*, May 1980.

[Che76] P. P. Chen, "The Entity-Relationship Model: Toward a Unified View of Data", *ACM Trans. Database Systems 1*, 1 (Mar. 1976).

[Dav84] M. Davis, "Automatic Layout of Graphs", M.S. Project Report, EECS Dept., U.C. Berkeley, Dec. 1984.

[For71] J. W. Forrester, *World Dynamics*, Wright-Allen, 1971.

[Goe85] K. J. Goldman and et. al., "ISIS: Interface for a Semantic Information System", *Proc. 1980 ACM SIGMOD Conf. on Mgt. of Data*, May 1985.

[HaO84] G. T. Hamachi and J. K. Ousterhout, "A Switchbox with Obstacle Avoidance", *Proc. IEEE 21st Design Automation Conf.*, 1984, 173-179.

[Her80] C. Herot, "SDMS: A Spatial Data Base System", *ACM Trans. on Database Systems*, Dec. 1980.

[HoT74] J. Hopcroft and R. Tarjan, "Efficient Planarity Testing", *J. ACM* , Oct. 1974, 549-568.

[Hsu83] C. P. Hsu, "Minimum-Via Topological Routing", *IEEE Trans. on CAD*, Oct. 1983, 235-246.

[Joy84] W. Joy, "An Introduction to Display Editing with Vi", *Unix User's Manual – Supplementary Documents*, Mar. 1984. U.C. Berkeley.

[Lie85] R. J. Lipton and et. al., "A Method for Drawing Graphs", *proc. who knows where*, May 1985.

[Mar85] J. Martin, SYN – A Language for Typesetting Syntax Charts, unpublished manuscript, Dept. Comp. Sci., Univ. of Minn., 1985.

[Mey83] C. Meyer, "A Browser for Directed Graphs", M.S. Project Report, EECS Dept., U.C. Berkeley, Dec. 1983.

[MiW84] C. Mills and A. Wasserman, "A Transition Diagram Editor", *Proc. 1984 Summer Usenix Meeting*, June 1984, 287-296.

[Ree77] E. M. Reingold and et. al., in *Combinatorial Algorithms, Theory and Practice*, Prentice Hall, Englewood Cliffs, NJ , 1977, 364-391.

[Row85] L. Rowe, "Grab Update Protocol (Version 1.0)", unpublished document, U.C. Berkeley, Dec. 1985.

[Sae84] A. Sangiovanni and et. al., "A New Gridless Channel Router: Yet Another Channel Router the Second (YACR-II)", *Proc. IEEE ICCAD-84*, 1984, 72-75.

[StK82] M. Stonebraker and J. Kalash, "TIMBER: A Sophisticated Relation Browser", *Proc. 8th Very Large Data Base Conference*, Sep. 1982.

[StR85] M. R. Stonebraker and L. A. Rowe, "The Design of POSTGRES", submitted for publication, Dec. 1985.

[STT81] K. Sugiyama, S. Tagawa and M. Toda, "Methods for Visual Understanding of Hierarchical System Structures", *IEEE Trans. on Sys. Man, and Cyb. SMC-11*, 2 (Feb. 1981), 109-125.

[Sug84] K. Sugiyama, "A Readability Requirement in Drawing Digraphs: Level Assignment and Edge Removal for Reducing the Total Length of Lines", Research Report No. 45, International Institute for Advanced Study of Social Information Science (IIAS-SIS), Fujitsu Ltd., March 1984.

[SUN84] *SUN II Workstation*, Sun Microsystems, Inc., Mountain View, CA, Jan. 1984.

[Tae83] R. Tamassia and et. al., "An Algorithm for Automatic Layout of Entity Relationship Diagrams", in *Entity-Relationship Approach to Software Engineering*, C. G. Davis and et. al. (editor), North Holland, 1983, 421-439.

[Tes81] L. Tessler, "The Smalltalk Envrionment", *Byte* , Aug. 1981.

[SCA83] *SCALD System Reference Manual*, Valid Logic Systems, Inc., San Jose, CA, 1983.

[RTI84] *INGRES QBF (Query By Forms) User's Guide*, Version 3.0, VAX/VMS, Relational Technology, Inc., Berkeley, CA, May 1984.

[Woo81] D. R. Woods, "Drawing Planar Graphs", PhD Thesis, Dept. of Comp. Sci., Stanford University, June 1981.

[Zlo75] M. M. Zloof, "Query by Example", *Proc. NCC 44* (1975).